

COURS
DE
FORTRAN
90

Matmeca 1ère année
ENSEIRB-MATMECA

Luc Mieussens

Luc.Mieussens@math.u-bordeaux1.fr

- équipe *Calcul Scientifique et Modélisation* -
Institut de Mathématiques de Bordeaux
Université de Bordeaux

version du 28 septembre 2011

Le but de ce cours est d'apprendre à programmer en fortran 90. Il est prévu pour 8 séances d'1h20, accompagné de 8 séances de quatre heures de TD-TP sur machine.

J'ai essayé de le présenter non pas comme une collection exhaustive de commandes (ce qui permet difficilement de pouvoir programmer avant d'avoir fini le cours) mais plutôt comme une progression en trois phases. La première partie devrait permettre de pouvoir écrire vos premiers programmes après une ou deux séances seulement. La deuxième introduit certains concepts plus complexes (sous-programmes, types dérivées, allocation dynamique). Les troisième et quatrième parties mettent l'accent sur les apports réellement nouveaux de fortran 90 par rapport au vieux fortran 77 (modules, interfaces génériques, surcharge d'opérateurs, pointeurs).

Nombre de commandes présentées ici sont expliquées avec beaucoup plus de précision dans divers cours et manuels de référence donnés dans la bibliographie. Ces références sont en accès libre et gratuit sur internet. N'hésitez pas à les consulter dès que vous en avez besoin.

Le langage fortran évolue régulièrement, mais la véritable évolution a été faite en passant du fortran 77 au fortran 90. Les nouvelles normes fortran 95 (puis fortran 2003, fortran 2008) n'apportent que des fonctionnalités nouvelles. Par conséquent, tout ce qui est dit ici devrait rester valable avec les normes futures.

Enfin, je veux insister sur le fait que quel que soit le cours que vous utiliserez, il n'y en fait qu'un seul moyen d'apprendre à programmer : c'est de programmer soi-même (de la même façon que l'on n'apprend bien une langue étrangère qu'en la parlant soi-même). Donc usez et abusez de votre temps en salle machine (le temps des séances de TP n'y suffira pas), c'est la garantie du succès.

Table des matières

1	Introduction	6
1.1	Qu'est-ce qu'un programme informatique ?	6
1.2	Le langage F90 : généralités	9
2	Premiers pas en F90	11
2.1	Exemple de programme	11
2.2	structure	11
2.3	Les variables	11
2.4	les identificateurs	14
2.5	Continuation de ligne : caractère &	15
2.6	Les commentaires : symbole !	16
2.7	Les structures de contrôle	17
2.7.1	Le test if	17
2.7.2	La sélection select case	20
2.7.3	Itérations : structure do	21
2.8	Les entrées-sorties écran/clavier	24
2.9	Compilation	25
2.10	Exécution	28
2.11	Ecrire un programme proprement	29
2.12	Exemple complet	30
3	Un peu plus loin ...	32
3.1	Variables “constantes”	32
3.2	Sous-programmes	33
3.2.1	Objectif	33
3.2.2	Fonctionnement	36
3.2.3	Arguments d'une subroutine	37
3.2.4	Arguments d'une fonction - déclaration du résultat	38
3.2.5	Les tableaux en arguments	40
3.2.6	Les chaînes en arguments	44

3.3	Tableaux	45
3.3.1	Tableaux statiques	45
3.3.2	Allocation dynamique	45
3.3.3	Allocation dynamique dans un sous-programme : tableaux automatiques	46
3.3.4	Terminologie	47
3.3.5	Opération conformes entre tableaux	48
3.3.6	Créer des tableaux “à la main”	49
3.3.7	Les sections de tableaux	51
3.3.8	Stockage des tableaux dans la mémoire et ordre des boucles	51
3.4	Entrées-sorties (E/S)	54
3.4.1	E/S en format libre, dans un fichier texte à accès séquentiel	54
3.4.2	E/S dans un fichier binaire à accès séquentiel	59
3.4.3	E/S formattées	61
3.4.4	Contrôle des erreurs en E/S	63
3.5	Trouver des erreurs dans un programme	65
3.5.1	Les éviter et les détecter	65
3.5.2	Erreurs détectées à la compilation	66
3.5.3	Erreurs détectées à l'exécution	68
3.5.4	Erreurs d'algorithme	69
3.5.5	Les “debuggers”	70
4	Programmation modulaire	71
4.1	Introduction	71
4.1.1	Qu'est-ce-que c'est ?	71
4.1.2	Intérêt	71
4.2	Les procédures internes	72
4.3	Les procédures externes	72
4.3.1	Pourquoi “externes?”	72
4.3.2	Utilisation : problème d'interface	73
4.4	Les modules	75
4.5	Compilation séparée	81

5 Utilisation avancée	89
5.1 Types dérivés	90
5.2 Précision des réels	92
5.3 Fonctions de fonctions	94
5.4 Interface générique	96
5.5 Création de nouveaux opérateurs	100
5.6 Ressources privées, publiques, semi-privées	105
5.7 Pointeurs	106
Références	107
Appendice : Liste de quelques fonctions intrinsèques usuelles	108

1 Introduction

1.1 Qu'est-ce qu'un programme informatique ?

- tout commence par un algorithme : suite d'opérations donnant un résultat final
exemple :

approximation de $\sqrt{2}$ par méthode de Newton

u_0 donné	calculer $u_{n+1} = \frac{u_n}{2} + \frac{1}{u_n}$, pour $n = 0$ à 10.
-------------	--

- ces opérations sont écrites dans un fichier, avec un langage informatique (pour nous, le fortran 90 (f90))
 - ce fichier constitue le programme ;
 - c'est une suite d'opérations sur des variables, et éventuellement des échanges de données entre utilisateur et ordinateur ;
 - les variables sont des noms correspondant à de la place mémoire dans l'ordinateur, utilisée pour stocker les données et les manipuler.

→ exemple : approximation de $\sqrt{2}$

```
program racine
```

```
implicit none
```

```
! --- variables
```

```
integer :: n
```

```
real :: u
```

```
! --- initialisation
```

```
u=1.0
```

```
! --- boucle
```

```
do n=1,10
```

```
    u=u/2.0+1.0/u
```

```
end do
```

```
! --- affichage
```

```
print *, 'approx. de sqrt(2) : ', u
```

```
end program racine
```

- ce programme n'est pas utilisable tel quel : il faut le rendre compréhensible par l'ordinateur, c.-à-d. le traduire : le logiciel traducteur est appelé *compilateur*, la traduction est appelée *compilation*.

exemple :

```
gfortran -o prog prog.f90
```

prog.f90 : fichier contenant le programme écrit en f90

prog : traduction (appelé *fichier exécutable*)

- le fichier exécutable permet d'exécuter les instruction programmées.

exemple :

```
./prog
```

1.2 Le langage F90 : généralités

- ensemble de règles syntaxiques et grammaticales et de mots permettant d'écrire des opérations mathématiques, de communiquer des informations entre l'ordinateur (mémoire), les fichiers (disque dur) et l'utilisateur (clavier/écran).
- l'essentiel du cours va porter sur l'apprentissage de ce langage :
 - en commençant par une vue rapide et simplifiée permettant de pouvoir programmer après une ou deux séances ;
 - puis en découvrant ensuite les aspects plus complexes du langage.
- conventions typographiques utilisées dans ce document : en **gras** pour les mots-clefs, entre crochets [] pour les instructions optionnelles, en *italique* pour les commandes unix, en **teletype** pour les lignes fortran.

- historique : premier compilateur (langage de haut niveau) en 1957, évolutions constantes depuis (voir wikipedia) :
 - 1978 : Fortran 77, très répandu aujourd’hui encore dans les gros codes de calculs
 - 1990 : Fortran 90, fortran devient un langage moderne (format libre, programmation modulaire)
 - fortran 95, 2000 (programmation orientée objet), 2004, ...
- F90 ou C++ pour le calcul scientifique : la majeure partie des gros codes de calcul industriel sont écrits en fortran ...
- *Attention aux effets de mode (“Fortran est dépassé”, “vous programmez encore en fortran ?”, “il faut programmer en C++”)* :
 - fortran est bien plus simple d'utilisation
 - les concepts de C++ ne sont guère utiles pour le calcul scientifique en général
 - les aspects programmation ”orientée objet” du C++ sont maintenant utilisables en fortran (depuis fortran 90 et fortran 2003)
 - de nouveaux compilateurs fortran sont développés pour les nouvelles architectures (fortran pour les GPU par exemple).
 - attention aux commentaires de ceux qui ne connaissent que le C (et de ceux qui ne connaissent que le fortran !)
 - voir le blog d'un physicien qui connaît bien les deux langages : http://qd.typepad.com/4/2005/08/c_vs_fortran.html

2 Premiers pas en F90

2.1 Exemple de programme

2.2 structure

```
program nom_du_programme  
    declaration des variables  
    instructions  
end program nom_du_programme
```

2.3 Les variables

- variables : noms (chaînes de caractères alphanumériques plus quelques autres, cf. section 2.4) permettant de manipuler des données en mémoire.
- opération essentielle : l'*affectation* (symbole =)
exemple : variable **n** (entier) :

```
n=2      ! -- prend la valeur 2  
n=n+1   ! -- augmente de 1
```

ce n'est pas une égalité mais une opération : la valeur de *n* stockée en mémoire est remplacée par elle-même plus 1.

- le bloc *déclaration des variables* sert à indiquer à la machine quel est le *type* de chaque variable utilisée afin de résERVER la place en mémoire nécessaire.
exemple :

déclaration	signification	ex. d'affectation
<code>integer :: n,i</code>	entier	<code>n=10</code>
<code>real :: x,y</code>	réel	<code>x=1.0e-5</code> <code>y=1.0</code>
<code>complex :: z</code>	complexe	<code>z=cmplx(1.0,2.5e3)</code>
<code>character :: c</code>	caractère	<code>c='d'</code>
<code>logical :: b</code>	booléen	<code>b=.true.</code>

- déclaration implicite : bien que f90 ne l'exige pas, il est prudent de déclarer toutes les variables. Pour éviter les oubliS, placer l'instruction **implicit none** au début du bloc déclaration. Les oubliS sont alors détectés à la compilation.

- les tableaux : tous les types précédents peuvent être structurés en tableaux avec l'attribut **dimension**

```
integer , dimension( -2:2 ) :: dim
real , dimension( 1:3 , 0:4 , 1:20 ) :: tab
character , dimension( 1:3 , 4:6 ) :: tch
```

→ chaque argument de **dimension** indique les bornes de la dimension correspondante par un intervalle entier `min:max`

→ accès aux composantes :

```
dim(0)=10
tab(2,3,20) = 2.3e-5
tch(1,5)='a'
```

2.4 les identificateurs

- caractères du clavier permettant de nommer une variable ou un autre élément du programme ;
- suite de 1 à 31 caractère alphanumériques parmi : lettres sans accents majuscules et minuscules, chiffres, caractère _ (souligné) ;
- le 1^{er} caractère doit être une lettre ;
- les majuscules et minuscules ne sont pas différenciées ;
- exemple :
 - les identificateurs suivants sont valides :

```
constante_gaz  
pi2  
Rk54
```

→ les identificateurs suivants ne sont pas valides :

```
accentué  
avec espace  
Il_y_a_plus_de_trente_et_un_caracteres  
_souligne_devant  
1_chiffre_devant  
nom#alphanumerique
```

2.5 Continuation de ligne : caractère &

- une ligne d'instructions comporte au maximum 132 caractères
- si plus de 132 \Rightarrow message d'erreur :

```
write(*,*) 'Au clair de la lune, mon ami Pierrot. Prete moi ta plume, pour ecrire un mot.  
Ma chandelle est morte, je n ai plus de feu. Ouvre moi la porte pour l amour de dieu'  
1
```

Error : Unterminated character constant beginning at (1)
- solution : couper la ligne en morceaux. Placer & en fin de première ligne et & au début de la ligne suivante :

```
write(*,*) 'Au clair de la lune, mon ami Pierrot. &  
&           Prete moi ta plume, pour ecrire un mot. &  
&           Ma chandelle est morte, je n ai plus de &  
&           feu. Ouvre moi la porte pour l amour de dieu'
```

2.6 Les commentaires : symbole !

- tout ce qui est à droite du symbole ! est ignoré par le compilateur
- cela permet d'écrire des commentaires dans le programme, c.-à-d. du texte qui en explique le fonctionnement
- le ! peut être placé n'importe où dans la ligne (au début, au milieu, à la fin), et à n'importe quel endroit du programme
- exemple : voir le programme page 30.

2.7 Les structures de contrôle

Ce sont des instructions qui permettent de faire des *itérations* ou des *tests* sur des variables.

2.7.1 Le test **if**

- exemple : calcul de la valeur $y = \begin{cases} x \ln x & \text{si } x \neq 0 \\ 0 & \text{sinon} \end{cases}$

```
if (x /= 0.0) then
    y = x * log(x)
else
    y = 0.0
end if
```

- syntaxe générale :

```
[nom:] if (expression logique) then
    instructions
[elseif (expression logique) then
    instructions]
[else
    instructions]
end if [nom]
```

où nom permet de donner un nom à la structure pour une meilleure lisibilité du programme (par exemple en cas de **if** imbriqués)

- **expression logique** est une combinaison d'un ou plusieurs tests sur des variables, ainsi que des trois opérateurs logiques

test	signification
<code>==</code>	égal
<code>/=</code>	différent
<code>></code>	supérieur strict
<code><</code>	inférieur strict
<code>>=</code>	supérieur ou égal
<code><=</code>	inférieur ou égal
<code>.and.</code>	et
<code>.or.</code>	ou
<code>.not.</code>	non

- les variables de type **logical** peuvent aussi être utilisées
- le **else if** et le **else** sont optionnels

- autre exemple : calcul de $y = \begin{cases} 0 & \text{si } x < 0 \text{ ou } x > 2 \\ \sqrt{x} & \text{si } x \in [0, 1] \\ 1 & \text{si } x > 1 \end{cases}$

```
rac: if (x<0.0.or.x>2.0) then
    y = 0.0
elseif (x<=1.0) then
    y=sqrt(x)
else
    y=1.0
end if rac
```

2.7.2 La sélection `select case`

- pour choisir une instruction à exécuter en fonction de la valeur d'une expression scalaire de type entier, caractère ou logique
- exemple :

```
character(len=30) :: pays
...
langue:select case(pays)
case( 'france' , 'quebec' , 'suisse' , 'belgique' )
    print*, 'bonjour'
case( 'royaume-uni' , 'usa' )
    print*, 'hello'
case default
    print*, 'langue pas disponible'
end select langue
```

- l'expression testée doit vérifier au plus un **case**
- le **case default** n'est pris en compte que si l'expression ne vérifie aucun **case**

2.7.3 Itérations : structure do

- forme énumérative (nombre d’itérations fixé)

exemple : calcul de $y = \sum_{i=1}^n \frac{x^i}{i}$

```
integer :: i, n
real :: x, y
...
y=0.0
do i=1,n
    y=y+x**i/i
end do
```

syntaxe :

```
[ nom :] do variable=debut , fin [ , pas ]
            instructions
end do [ nom ]
```

autre exemple : calcul de $s = 1 + 3 + 5 + 7 + 9$

```
integer :: i, s
...
s=0
do i=1,9,2
    s=s+i
end do
```

- forme infinie : on ne sort qu'avec l'instruction **exit**

```
[ nom : ] do
    instructions
    if ( condition ) then
        exit [ nom ]
    end if
    instructions
end do [ nom ]
```

exemple :

```
do
    read * , nombre
    if ( nombre==0) then
        exit
    end if
    somme = somme + nombre
end do
```

- forme conditionnelle : arrêt si la condition spécifiée en début de boucle est satisfaite

```
[nom:] do while (condition)
        instructions
end do [nom]
```

exemple :

```
reponse='oui'
lecture:do while (reponse=='oui')
    print*, 'continuer ?'
    read*,reponse
end do lecture
```

2.8 Les entrées-sorties écran/clavier

- lecture au clavier de la valeur d'une variable, quel que soit son type
read*,variable
- écriture à l'écran d'un message et/ou de la valeur d'une variable
print*, 'bonjour'
print*,x
print*, ' il est ',h, 'heures'
- exemple :

```
integer :: n
character(len=15) :: pays
...
print*, 'quel age avez-vous ? '
read*,n
print*, 'vous avez ',n,'ans'
print*, 'de quel pays etes vous ? '
read*,pays
print*, 'vous venez de ',pays
```

2.9 Compilation

- pour faire fonctionner un programme écrit en f90, il faut le traduire en un langage compréhensible par l'ordinateur : c'est la *compilation*.
- cette compilation est effectuée par un *compilateur*. Dans ce cours, nous utiliserons le compilateur libre *gfortran*.
- la compilation crée un fichier dit *exécutable*.
- exemple : le programme s'appelle **calcul**, il est écrit dans le fichier *toto.f90*. On souhaite compiler ce programme pour créer un exécutable nommé ici *compute*. On tape donc la commande

```
gfortran -o compute toto.f90
```

- plus généralement, la commande de compilation est :

```
gfortran -o executable fichier_fortran
```

où *executable* est le nom que l'on souhaite donner au fichier exécutable et *fichier_fortran* est le nom du fichier qui contient le programme f90 (il est conseillé de donner au fichier le même nom que le programme, en rajoutant l'extension *.f90*).
- différentes options existent, cf. section 3.

- messages d’erreurs :
 - le compilateur détecte certaines erreurs (syntaxe, conformité de types, de dimensions)
 - il affiche un message donnant le numéro de la ligne contenant l’erreur, avec une brève explication en anglais
 - exemple : (programme contenu dans le fichier erreur-exemple.f90)

```
erreur_exemple.f90:7.3:
```

```
y = x*log(x)
    1
Error: Symbol 'y' at (1) has no IMPLICIT type
```

- une erreur en provoque souvent d’autres (exemple : une variable non déclarée va générer des erreurs à chaque ligne où elle est utilisée)
- la liste des erreurs est donc souvent très longue, et il faut alors commencer par corriger les premières de la liste, puis recompiler.

- les messages d'erreurs ne sont pas toujours très clairs ...

exemple : dans un programme contenu dans le fichier erreur_declaration.f90, on fait une erreur de frappe dans la déclaration suivante :

ineger :: n

au lieu de :

integer :: n

ce qui provoque le message suivant :

```
erreur_declaration.f90:5.2:
```

```
    ineger :: n
    1
Error: Unclassifiable statement at (1)
```

2.10 Exécution

- si le fichier contenant l'exécutable s'appelle `prog`, alors taper simplement `./prog`
- erreurs à l'exécution
 - toutes les erreurs ne sont pas détectées à la compilation (certaines options peuvent cependant y aider)
 - divisions par 0, erreurs d'affectation de tableau
 - erreurs d'algorithme (+ au lieu de -)
 - il faudra apprendre à trouver ces erreurs en faisant des tests, et en utilisant un debugger (cf. sec. 3.5.5).

2.11 Ecrire un programme proprement

- écrire des commentaires (but et fonctionnement du programme, signification des variables, décrire chaque bloc d'instruction)
- indenter correctement (en particulier les structures de contrôle)
- nommer les boucles imbriquées
- déclarer les variables dans l'ordre croissant des structures (logiques, entiers, caractères, réels, puis tableaux)
- mieux vaut utiliser des noms de variables longs mais clairs que des noms courts et obscurs
- il est fortement conseillé d'utiliser l'éditeur de texte **Emacs** pour programmer. Outre sa puissance d'édition en général (au moyen de raccourcis clavier qui permettent de taper du texte très rapidement), il possède des fonctions adaptées à la programmation : indentation automatique, complétion des noms de programmes et de boucles, coloration des mots clefs, etc. Voir le guide Emacs distribué en cours [5].

2.12 Exemple complet

calcul de la solution de $x - e^{-x} = 0$ par la méthode de Newton.

algorithme :

$$\begin{aligned} & x_0 \text{ et } tol \text{ donnés} \\ & \text{calculer } x_{n+1} = x_n - \frac{(x_n - e^{-x_n})}{1 + e^{-x_n}} \\ & \text{tant que } \left| \frac{x_{n+1} - x_n}{x_n} \right| \geq tol \end{aligned}$$

```
program zero

implicit none

! --- variables
real :: x,y,diff,tol

! --- lecture des données
print *, 'entrer le x initial'
read *, x

print *, 'tolérance '
read *, tol

! --- méthode de newton
newton: do

    ! -- itération
    y=x-(x-exp(-x))/(1.0+exp(-x))

    ! -- différence entre deux itérés
```

```

if (x/=0.0) then
    diff=abs((x-y)/x)
else
    diff=abs(x-y)
end if

!-- mise à jour
x=y

!-- test de convergence
if (diff<tol) then
    exit
end if

end do newton

print *, 'solution approchée=' , x

end program zero

```

3 Un peu plus loin ...

3.1 Variables “constantes”

Pour déclarer une variable “constante” (c.-à-d. qui a une valeur non modifiable) :

- attribut **parameter**
- la variable est non modifiable, sinon erreur détectée à la compilation
- utile pour déclarer et protéger des constantes physiques
- exemple :
real, parameter :: pi=3.14159265, N=6.02252e23
- dans la déclaration, les opérations usuelles sont utilisables, mais pas l’appel de fonction :
real, parameter :: quart=1.0/4.0 !--- valide
real, parameter :: pi=acos(-1.0) !--- non valide

3.2 Sous-programmes

3.2.1 Objectif

- bloc d’instructions utilisé plusieurs fois ⇒ l’écrire dans un sous-programme une fois pour toutes, et l’appeler quand nécessaire (aussi appelé procédure)
- deux types de sous-programme : les **subroutines** et les **fonctions**

- exemple 1 : subroutine

```
program exproc

implicit none

real :: moyenne, maximum
real, dimension(100) :: tab

call random_number(tab)
call sp( tab, moyenne, maximum )
print *, moyenne, maximum

contains

subroutine sp( t, moy, max )

implicit none

real, dimension(100), intent(in) :: t
real, intent(out) :: moy, max
integer :: i

max = t(1); moy = t(1)
do i=2,100
    if (t(i) > max) then
        max = t(i)
    end if
    moy = moy + t(i)
end do
moy = moy / 100

end subroutine sp

end program exproc
```

- exemple 2 : fonction

```
program exfct

implicit none

real :: maximum
real , dimension(100) :: tab

call random_number(tab)

maximum = maxi(tab)
print *, maximum

contains

function maxi(t)

implicit none

real , dimension(100) , intent(in) :: t
integer :: i
real :: maxi ! - la fonction est déclarée

maxi = t(1)
do i=2,100
    if ( t(i) > maxi ) then
        maxi = t(i)
    endif
end do
end function maxi

end program exfct
```

3.2.2 Fonctionnement

- place dans le programme :
avant la dernière ligne **end program** nom_programme
et après le mot-clef **contains**

```
program nom_programme
    declarations
    instructions
    contains
        subroutine nom_subroutine
        ...
        end subroutine nom_subroutine

        function nom_fonction
        ...
        end function nom_fonction
end program nom_programme
```

- on parle alors de sous-programme *interne* (par opposition aux sous-programmes externes, cf. section 4.3)
- portée des variables : toutes les variables du programme principal sont connues de ses sous-programmes internes. Pas besoin de redéclaration, sauf pour les arguments (voir plus loin).
- les variables utilisées seulement par un sous-programme (*variables locales*) : déclarées à l'intérieur de celui-ci, comme dans un programme principal. Connues de lui seul.

- appel d'un sous-programme :
 - subroutine : appelée par l'instruction **call**

```
call nom_subroutine( arguments )
```

- fonction : appelée comme une fonction mathématique :

```
y=nom_fonction( arguments )
```

3.2.3 Arguments d'une subroutine

subroutine : sous-programme ayant ou non des arguments d'entrée et de sortie :

- variables à déclarer à l'intérieur de la subroutine
- il est conseillé de préciser dans ces déclarations les attributs suivants

intent(in) pour les arguments d'entrée

intent(out) pour les arguments de sortie

intent(inout) pour les arguments mixtes. Ainsi le compilateur détecte une mauvaise utilisation des arguments (entrée modifiée, sortie non donnée, etc.)

3.2.4 Arguments d'une fonction - déclaration du résultat

fonction : comme une subroutine, mais

- arguments d'entrée obligatoires
- renvoie nécessairement un résultat, stocké dans une variable ayant le même nom que la fonction

- le type du résultat peut être donné soit avant le nom de la fonction (sauf si résultat tableau) :

```
real function maxi(t)
  ...
end function maxi
```

soit dans une déclaration :

```
function maxi(t)
  ...
  real :: maxi
  ...
end function maxi
```

- le résultat peut avoir un nom différent du nom de la fonction, en utilisant la clause **result**

```
function maxi(t) result(y)
  ...
  real :: y
  ...
end function maxi
```

dans ce cas, pas de type avant le mot clef **function**.

- la variable résultat ne doit pas avoir d'attribut **intent**

3.2.5 Les tableaux en arguments

- problème : dans l'exemple de la page 34, la subroutine `sp` n'accepte que des tableaux de taille 100. Comment faire pour qu'elle fonctionne avec des tableaux de tailles quelconques ?
- le plus simple : donner en entrée le tableau et ses dimensions ("assumed size array" en anglais) :

```
program exproc
  ...
  real , dimension(100) :: tab
  ...
contains
  subroutine sp( t , n , moy , max )
    ...
    integer , intent(in) :: n
    real , dimension(1:n) , intent(in) :: t
    real , intent(out) :: moy , max
    integer :: i

    max = t(1); moy = t(1)
    do i=2,n
      ...
    end do
    moy = moy/100

  end subroutine sp

end program exproc
```

- autre exemple :

```
program toto
  ...
  real , dimension( 1:3 , 1:2 ) :: A
  ...
  call sousprog( A , 3 , 2 )
  ...
contains
  subroutine sousprog( A , m , n )
    integer , intent( in ) :: m , n
    real , dimension( 1:m , 1:n ) :: A
    ...
  end subroutine sousprog
end program toto
```

- on peut aussi ne passer que le tableau en argument et récupérer ensuite ses dimensions (tableau à *profil implicite*, "assumed shape array" en anglais). Attention alors à la déclaration :

```

program exproc
  ...
  real , dimension(100) :: tab
  ...
contains
  subroutine sp( t , moy , max )
    ...
    real , dimension(:) , intent(in) :: t
    real , intent(out) :: moy , max
    integer :: i,n

    n=size(t)
    max = t(1); moy = t(1)
    do i=2,n
      ...
    end do
    moy = moy/100

  end subroutine sp

end program exproc

```

- autre exemple :

```

program toto
  ...
  real , dimension(1:3,1:2) :: A,B
  real , dimension(1:3) :: u,v
  ...
  call sousprog(A,B)
  ...
  v=fct(u)
  ...
  contains
    subroutine sousprog(A,B)
      real , dimension(:,:) , intent(in) :: A
      real , dimension(:,:) , intent(out) :: B
      integer :: m,n
      m=size(A,1) ; n=size(A,2)
      B=A**2
    end subroutine sousprog

    function fct(u) result(v)
      real , dimension(:) , intent(in) :: u
      real , dimension(size(u)) :: v
      ...
    end function fct
  end program toto

```

- ATTENTION : le profil implicite s'utilise seulement pour les tableaux arguments. Pour des tableaux locaux ou un résultat de fonction dont les tailles dépendent de celles des arguments, remarquer l'utilisation de l'instruction **size**.

3.2.6 Les chaînes en arguments

même principe que pour les tableaux :

- on peut passer en argument la chaîne et sa longueur
- on peut aussi utiliser un profil implicite, pour les arguments uniquement
- exemple :

```
program chaine
  ...
  character(len=20) :: ch

  print *, ch, f2(ch, 20)
  print *, f1(ch)

contains

  function f1(ch)
    implicit none
    character(len=*), intent(in) :: ch
    character(len=len(ch))) :: f1
    ...
  end function eff_blanc

  function f2(ch, n)
    implicit none
    integer, intent(in) :: n
    character(len=n), intent(in) :: ch
    integer :: f2
    ...
  end function nb_blancs

end program chaine
```

3.3 Tableaux

3.3.1 Tableaux statiques

déjà vu : les dimensions sont fixées à la déclaration :

```
real , dimension(1:10,1:4) :: A
```

3.3.2 Allocation dynamique

- il est fréquent que les tailles des tableaux utilisés dans un programme dépendent de données et ne soient pas connues à la compilation
- moyen simple et efficace : attribut **allocatable** (tableau allouable) :

```
real , dimension (: ,:) , allocatable :: A
```

- le nombre de dimensions est donné (ici 2), mais pas les dimensions elles-mêmes
- pour manipuler ces tableaux, on commence par les *allouer* en mémoire (c.-à-d. la place nécessaire est réservée), en donnant leurs dimensions. Exemple :

```
integer :: n , m  
...  
read *,n ,m  
allocate (A (1:n ,1:m ))
```

- pour changer la taille du tableau : le désallouer, puis le réallouer :

```
deallocate(A)
allocate(A(1:3,0:4))
```

- évidemment, on ne peut utiliser un tableau non encore alloué.
- ne pas oublier de libérer la place mémoire avec **deallocate** quand le tableau n'est plus utilisé (en particulier pour les tableaux locaux dans des sous-programmes, sinon, risque de dépassement de mémoire).

3.3.3 Allocation dynamique dans un sous-programme : tableaux automatiques

- autre moyen simple de créer un tableau dont la taille n'est pas connue à la compilation
- utilisable seulement au sein d'un sous-programme
- principe : les dimensions du tableau sont des variables passées en arguments du sous-programme ("automatic array" en anglais). Exemple :

```
subroutine autom(m,n)
...
integer :: m,n
real, dimension(m,n) :: A
```

- avantage : plus rapide en exécution que la création d'un tableau allouable. A utiliser pour les sous-programmes appelés très fréquemment. Les instructions fortran sont aussi plus simples à écrire.
- inconvénient : aucun moyen de vérifier s'il y a assez de place pour créer le tableau, contrairement aux tableaux allouables. Le message d'erreur correspondant est généralement très obscur.
- pour les tableaux de grandes tailles utilisés peu fréquemment, il est donc plutôt conseillé d'utiliser les tableaux allouables.

3.3.4 Terminologie

- *rang* : nombre de dimensions (le rang de A est 2)
- *taille* d'une dimension : nombre d'éléments dans cette dimension (les tailles des dimensions de A sont 3 et 5)
- *forme* : suite ordonnée des tailles des dimensions (la forme de A est 3,5)
- une taille peut être nulle

3.3.5 Opération conformes entre tableaux

- but : faire des opérations sur des tableaux sans écrire les opérations élément par élément
- deux tableaux sont conformes s'ils ont même forme :
 - A (1:10,1:4) et B (3:12,0:3) sont conformes
 - A (1:10,1:4) et B (3:12,1:3) ne sont pas conformes
 - A (1:1,1:1) et B(1:1) ne sont pas conformes
- une expression est conforme si elle fait intervenir des tableaux conformes
- un scalaire est considéré comme conforme à tout tableau
- la plupart des opérations scalaires sont applicables aux tableaux : opération appliquée élément par élément, résultat de même forme que les tableaux opérandes

– exemples :

```
program conforme

implicit none

integer :: i,j
real , dimension(3,3) :: A,B,C

call random_number(A)
call random_number(B)

C=A+B+2.0          ! -- op. équivalente à
print*,C
do i=1,3
  do j=1,3
    C(i,j)=A(i,j)+B(i,j)+2.0
  end do
end do
print*,C

C=A*B              ! -- pas prod. matriciel
print*,C
C=sqrt(B)*C
print*,C
C=B+cos(B*3.1416)
print*,C
A=1-sin(log(B+A))
print*,A

end program conforme
```

3.3.6 Créer des tableaux “à la main”

- avec une liste de variables d'un même type `var1, var2 ,..., varn`, on peut créer un vecteur (tableau de rang 1) avec le constructeur `(/.../)` :

```
v=(/var1 ,var2 ,... ,varn /)
```

- la liste peut-elle même être créée avec une *boucle implicite*

```
v=(((i)**2,i=1,4))           ! --équivaut à  
v=(/1,4,9,16/)
```

- on peut imbriquer plusieurs boucles implicites

```
v=((((i+j,j=1,2),i=1,3))  ! --équivaut à  
v=(/2,3,3,4,4,5/)
```

- comme dans une boucle **do** on peut donner un pas :

```
v=/(i,i=1,11,2)/)      ! -- équivaut à
v=(1,3,5,7,9,11/)
```

- pour transformer un vecteur en tableau à plusieurs dimensions :

→ fonction **reshape(vecteur,forme_du_tableau)**

```
real , dimension(2,3) :: A
A=reshape(v,(/2,3/))
```

donne la matrice $A = \begin{pmatrix} 1 & 5 & 9 \\ 3 & 7 & 11 \end{pmatrix}$ (remplie *colonne par colonne*)

3.3.7 Les sections de tableaux

on accède à une section de tableau en modifiant convenablement les indices de début et de fin des dimensions :

```
real , dimension(1:10,1:20) :: A
A(2:6,1:20)      ! -- lignes de 2 à 6
A(2:6,:)         ! -- idem
A(i,: )           ! -- ième ligne
A(:,j)            ! -- jème colonne
A(2:6,1:4)        ! -- élément communs aux
                  !   lignes 2 à 6 et
                  !   colonnes 1 à 4
```

3.3.8 Stockage des tableaux dans la mémoire et ordre des boucles

- il est très important lorsqu'on manipule de gros tableaux de savoir comment ils sont stockés en mémoire
- exemple : y-a-t'il une différence importante entre les deux blocs suivants ?

```
do i=1,m  
  do j=1,n  
    c(i,j)=a(i,j)+b(i,j)  
  enddo  
enddo
```

```
do j=1,n  
  do i=1,m  
    c(i,j)=a(i,j)+b(i,j)  
  enddo  
enddo
```

⇒ oui : le premier bloc est beaucoup plus coûteux en temps !

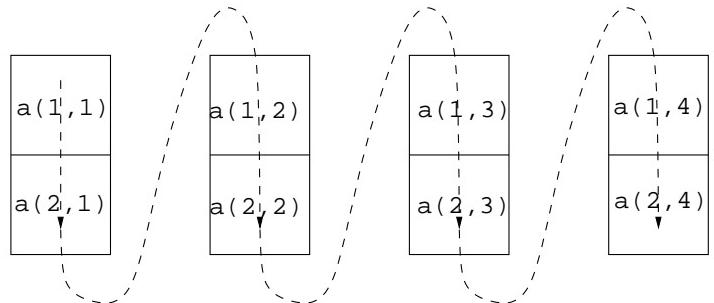
- quel que soit le nombre de dimensions d'un tableau, il est stocké en mémoire sous forme *uni-dimensionnelle*. Les éléments sont stockés ainsi :

$a(1,1)$	$a(2,1)$	$a(3,1)$	\dots	$a(m,1)$	$a(1,2)$	$a(2,2)$	$a(3,2)$	\dots
\dots	$a(1,n)$	$a(2,n)$	$a(3,n)$	\dots	$a(m,n)$			

- exemple : les éléments du tableau suivant
- integer, dimension(2,4) :: a**
sont rangés ainsi :

$a(1,1) \ a(2,1) \ a(1,2) \ a(2,2) \ a(1,3) \ a(2,3) \ a(1,4) \ a(2,4)$

$a(1,1)$	$a(1,2)$	$a(1,3)$	$a(1,4)$
$a(2,1)$	$a(2,2)$	$a(2,3)$	$a(2,4)$



- conclusion : quand on parcourt les éléments d'un tableau, il faut que la boucle sur la première dimension soit la boucle la plus interne (ceci est lié à la notion de *cache* dans la mémoire)
- c'est l'inverse de ce qui se fait en C, et de ce qui se fait en général quand on écrit les algorithmes
- voir en TP pour une comparaison en temps calcul des deux stratégies.

3.4 Entrées-sorties (E/S)

On a vu dans la première partie comment lire au clavier et écrire à l'écran. Ici, on va voir comment lire et écrire dans un fichier.

3.4.1 E/S en format libre, dans un fichier texte à accès séquentiel

C'est le plus simple, équivalent de ce qu'on a vu avec l'écran/clavier

- *ouverture* du fichier `toto` :

```
open ( unit=10, file='toto' )
```

- cette instruction ouvre le fichier à la première ligne (s'il n'existe pas, il est créé).
- l'expression `unit=10` indique que le fichier `toto` est connecté au descripteur numéro 10.
- tous les nombres peuvent être utilisés comme descripteurs sauf 5 (clavier) et 6 (écran).
- si le mot clef `file` n'est pas donné, le fichier `fort.n` est créé, où `n` est le descripteur donné après le mot clef `unit`.

- *écriture* de données dans `toto` (si associé au descripteur 10) :

```
write(10,*) donnee1, donnee2, donnee3
```

- une instruction **write** écrit toutes les données sur la même ligne, le **write** suivant passera à la ligne suivante.
- les données peuvent être de n'importe quel type (constantes ou variables)
- le **print***, pour écrire à l'écran est équivalent à `write(6,*)` et à `write(*,*)`

- *lecture* de données dans `toto` (si associé au descripteur 10) :

```
read(10,*) var1, var2, var3
```

- la première ligne de `toto` est lue, le **read** suivant passera à la ligne suivante.
- les données sont supposées être séparées par des blancs.
- elles sont stockées respectivement dans `var1`, `var2`, ..., `varn`.
- si plus de données que de variables : seules les premières données sont stockées (cas inverse : seules les premières variables sont affectées)
- le **read***, pour lire au clavier est équivalent à **read(5,*)** et à **read(*,*)**.
- *fermeture* du fichier `toto` :

```
close(10)
```

- attention, si on réouvre le fichier avec `open`, on se retrouve à la première ligne.
- écrasement de fichier : un **write** après l’ouverture d’un fichier existant détruit le contenu du fichier.

- exemple 1 :
- fichier titi.txt :

```
bonjour 4    heures
temperature   37.2
```

programme :

```
character(len=14) :: c1,c2,c3
integer :: n
real :: T

open(unit=10,file='titi.txt')
read(10,*) c1,n,c2      -- lecture 1ere ligne
read(10,*) c3,T          --           2eme ligne
close(10)

open(unit=20,file='toto.txt')

!-- ecriture 1ere ligne
write(20,*) c1,'il est ',n,c2

!-- ecriture 2eme ligne
write(20,*) 'la ',c3,'est de ',T,'degres'

close(20)
```

après exécution, fichier toto.txt :

```
bonjour           il est           4 heures
la temperature   est de       37.20000     degres
```

- exemple 2 : dans le fichier info.txt, liste de 1000 personnes avec un nom, age et taille sur chaque ligne.

Programme de lecture :

```
integer :: i
character(len=30), dimension(1:1000) :: nom
integer, dimension(1:1000) :: age
real, dimension(1:1000) :: taille

open(unit=30,file='info.txt')
do i=1,1000
  read(30,*) nom(i), age(i), taille(i)
end do
close(30)
```

3.4.2 E/S dans un fichier binaire à accès séquentiel

- les fichiers de textes sont des fichiers de caractères, ce qui peut prendre beaucoup de place s'il y a beaucoup de données.
- dans ce cas, il vaut mieux stocker les données (en particulier numériques) sous forme *binaire*. Le fichier n'est pas lisible avec un éditeur de texte, mais il est beaucoup plus petit.
- *ouverture* : avec **open** comme pour un fichier texte en précisant le paramètre **form='unformatted'** :

```
open(unit=15, file='toto', form='unformatted')
```

- *lecture/écriture* : **open** et **write** sans le paramètre * :

```
read(15) var1, var2  
write(15) donnee
```

- exemple : tableau de réels aléatoires, stocké dans un fichier texte et dans un fichier binaire. Comparer les tailles des deux fichiers.

```
real , dimension(1:1000) :: tableau

call random_number(tableau)

open(unit=1,file='texte.txt')
write(1,*) tableau
close(1)

open(unit=2,file='binaire.b',form='unformatted')
write(2) tableau
close(2)
```

3.4.3 E/S formattées

- inconvénient de E/S en format libre :
 - fichier pas propre : réels écrits sous forme décimale si $> 10^{-2}$, sous forme scientifique sinon (cf le fichier `texte.txt` de la section 3.4.2).
 - lecture peu sûre : risque de mélange des variables
- *format* : on indique précisément le nombre et le type des variables à lire et écrire
- paramètre **fmt** de **read** et **write** :

```
read(descripteur, fmt='(chaine)') var1, ...  
write(descripteur, fmt='(chaine)') var1, ...
```

où `chaine` indique le type des données dans l'ordre où elles sont lues ou écrites

- formats de base (plus de détails sur [3]) :
 - `fmt='(nIm)'` : n entiers de m chiffres
 - `fmt='(nFm.d)'` : n réels sous forme décimale, formés de m caractères (dont le .), dont d après la virgule
 - `fmt='(nEm.d)'` : n réels sous forme scientifique, formés de m caractères, dont d après la virgule (dont le .)
 - `fmt='(nAm)'` : n chaînes de m caractères

Dans tous les cas, c'est le caractère espace qui sépare les n champs.

- exemple :

```

character(len=10) :: prenom, nom
integer :: age

prenom='JEAN '
nom='DURAND'
age=20

write(* ,fmt='(2A10,1I3)') prenom, nom, age

```

va écrire :

```

JEAN      DURAND      20
ALAIN      MARTIN      33

```

- si on doit utiliser plusieurs fois un même format, on peut stocker la chaîne associée au paramètre **fmt** dans une variable. Exemple :

```

formate='(2A10,1I3)'
write(* ,fmt=formate) prenom, nom, age

```

3.4.4 Contrôle des erreurs en E/S

- avec les instructions précédentes, le programme s'arrête à la moindre erreur (lecture dans fichier inexistant, lecture dépassant la taille du fichier, écriture texte dans un fichier binaire, etc.)
- des paramètres des commandes **open**, **read** et **write** permettent d'éviter l'arrêt et de déterminer la source de l'erreur.
- exemple : le paramètre **iostat** :

```
integer :: erreur
```

```
read(15,*,& iostat=erreur) ! - fichier texte
read(23,& iostat=erreur) ! - fichier binaire
```

- si lecture correcte \Rightarrow erreur vaut 0
- sinon, si erreur < 0 alors fin de fichier rencontrée, sinon autre erreur.

→ exemple :

```
erreur=0
do while(erreur==0)
    read(15,* ,iostat=erreur) var
    print* , 'variable=' ,var
end do

if (erreur<0) then
    print* , 'fin du fichier '
else
    print* , 'erreur de lecture '
end if
```

→ voir les instructions **open** et **write** pour la signification de iostat.

3.5 Trouver des erreurs dans un programme

3.5.1 Les éviter et les détecter

1. avant de programmer un algorithme, s'assurer qu'il est correct
2. programmer proprement (voir sec. 2.11), utiliser au maximum les signaux d'erreurs renvoyés par les commandes (paramètre **iostat**, etc.), afficher des messages d'erreur en cas d'opération interdite ($\sqrt{x} < 0$, etc.)
3. lors de la phase de développement, utiliser des options de compilation pour détecter les erreurs à l'exécution comme `-fcheck=all` qui détecte en particulier les dépassemens de tableau) ou `-ffpe-trap=invalid,zero` qui détecte les opérations invalides
4. effectuer des tests judicieux pour tester le programme
5. en désespoir de cause, rajouter des affichages de variables dans le programme, et utiliser l'instruction **stop** pour que le programme s'arrête à l'endroit voulu.

La tâche 5 peut être simplifiée en utilisant un “debugger” (voir sec. 3.5.5).

3.5.2 Erreurs détectées à la compilation

- le compilateur renvoie un message plus ou moins explicite (voir sec. 2.9)
- erreur de syntaxe : exemple la compilation sur natty de

```
program erreur_syntaxe
    implicit none

    integer : x
    real :: r

    r=10^2
end program erreur_syntaxe
```

renvoie les messages suivants :

```
erreur_syntaxe.f90:4.11:
    integer : x
        1
Error: Invalid character in name at (1)
erreur_syntaxe.f90:7.2:
    r=10^2
        1
Error: Unclassifiable statement at (1)
```

- oubli de déclaration : voir l'exemple précédent
- erreur de dimension : la compilation sur natty de

```
program erreur_dimension
implicit none

real , dimension(1:4) :: a

a(4,4)=1.0
end program erreur_dimension
```

renvoie les messages suivants

```
erreur_dimension.f90:6.3:
a(4,4)=1.0
          1
Error: Rank mismatch in array reference at (1) (2/1)
```

- malheureusement, il existe beaucoup d'autres erreurs possibles ...

3.5.3 Erreurs détectées à l'exécution

Il y a évidemment beaucoup d'erreurs possibles. En voici quelques unes avec le programme suivant

```
program erreur_execution
    implicit none
    real :: x

    x=1/3
    print *,x

    x=-1.0
    print *,sqrt(x)

    read *,x
end program erreur_execution
```

En supposant que l'utilisateur tape la lettre **a** au clavier, on aura alors les affichages suivants à l'écran :

```
0.0000000
NaN
a
At line 11 of file erreur_execution.f90 (unit = 5, file = 'stdin')
Fortran runtime error: Bad real number in item 1 of list input
```

- le premier print renvoie 0, car la division est entière (oubli de .0 sur le numérateur et/ou dénominateur). Mais il n'y a pas d'erreur d'exécution à proprement parler, puisque le programme continue à tourner)
- le deuxième renvoie NaN (not a number), car la racine d'un nombre négatif n'est pas définie (là non plus, pas vraiment de message d'erreur : le programme continue à tourner)
- le dernier affichage est un message d'erreur : le programme s'arrête (on dit qu'il “plante”) à cause de la lecture d'un caractère au lieu du réel attendu.

3.5.4 Erreurs d'algorithme

Ce sont des erreurs souvent difficiles à trouver. C'est pourquoi il vaut mieux bien écrire son algorithme avant de le programmer

3.5.5 Les “debuggers”

La tâche 4 décrite à la section 3.5.1 peut être simplifiée avec les logiciels dits “debuggers” :

- sans eux, il faut recompiler et exécuter le programme après chaque modification, ce qui peut être long quand le programme est gros ;
- un debugger permet, sans modifier le programme, d'afficher la valeur des variables au cours de l'exécution, et de stopper temporairement l'exécution à l'endroit désiré.
- l'utilisation d'un debugger nécessite un certain investissement, mais cet effort est largement récompensé par la suite.

Sur la machine natty, le debugger est *gdb*. Il peut être utilisé avec l'interface graphique *ddd* (voir le manuel [2] avec un exemple simple et instructif dans la section “sample session”).

Important : pour utiliser un debugger, il faut compiler le programme avec l'option `-g`.

4 Programmation modulaire

4.1 Introduction

4.1.1 Qu'est-ce-que c'est ?

- programmer de façon modulaire consiste à découper le plus possible un programme en morceaux algorithmiquement indépendants (appelés *sous-programmes*)
- exemple : calculer $t = \text{trace}(A^{-1}B)$. Ce problème contient trois calculs qui peuvent être programmés indépendamment :

$$A \mapsto \text{trace}(A)$$

$$(A, B) \mapsto AB$$

$$A \mapsto A^{-1}$$

on essaiera donc de rendre ces morceaux les plus indépendants possible au niveau informatique.

4.1.2 Intérêt

- permet de réutiliser plusieurs fois un morceau de programme facilement
- le programme est plus lisible
- le risque d'erreurs de programmation est moins grand : chaque bloc est écrit, compilé, et testé séparément.

- exemple : création d'un ensemble de fonctions pour faire des calculs matriciels (produit matrice-vecteur, triangulation, solution d'un système linéaire, déterminant, trace, etc.)
- on a déjà vu un moyen de créer des sous-programmes en utilisant les *fonctions* et *subroutines*. On va aussi voir les *modules*.

4.2 Les procédures internes

- déjà vu à la section 3.2, facile à faire
- problème : pas emboîtable. Une procédure interne ne peut pas contenir de sous-programme. Or une procédure interne peut nécessiter l'utilisation d'une autre procédure interne (exemple : la procédure de résolution d'un système linéaire par méthode de Gauss nécessite une procédure interne de résolution d'un système triangulaire)
- problème de lisibilité : le programme principal peut être très gros car il contient tous les sous-programmes
- cette méthode ne répond donc pas parfaitement au problème.

4.3 Les procédures externes

4.3.1 Pourquoi “externes”?

Ces procédures sont écrites à l'extérieur du programme principal :

- soit après l'instruction **end program** qui termine le programme principal
- soit dans un autre fichier

4.3.2 Utilisation : problème d'interface

Elles peuvent s'utiliser exactement comme une procédure interne (même syntaxe), mais il existe un problème d'interface :

- programme principal et procédures sont compilés séparément
- il n'y a pas de contrôle de cohérence des arguments entre l'*appel* et la procédure appelée : les erreurs éventuelles ne sont donc pas détectées à la compilation. Elles seront “visibles” à l'exécution, mais seront peut être difficiles à localiser. Exemple : types des variables différents entre l'appel et la procédure appelée.

- on dit que l'*interface* entre programme principal et procédure appelée est *implicite* (contrairement aux procédures internes où l'interface est dite explicite).
 - conséquence : il est impossible de passer des tableaux ou des chaînes à profils implicites dans une procédure externe avec interface implicite.
- ⇒ pour rendre une interface explicite, le plus simple est d'enfermer la procédure externe dans un **module** (il existe d'autres solutions moins simples avec les bloc **interface**, voir [3]).

4.4 Les modules

- utilisés pour que l’interface entre programme principal et procédure externe soit explicite
- on enferme la procédure externe dans un module de la façon suivante :

```
module nom_du_module

    implicit none

contains

    subroutine nom_subroutine

        ...

    end subroutine nom_subroutine

end module nom_du_module
```

- le module est alors placé dans un fichier (différent de celui du programme principal) qui a pour nom *nom_du_module.f90*
- dans le programme utilisant la subroutine, il suffit de rajouter l’instruction **use nom_du_module** en tout début de programme, avant l’instruction **implicit none**.

- exemple : erreur d’argument dans l’appel d’une procédure externe

1. interface implicite (sans module)
fichier *test1.f90*

```
program test1

implicit none

call somme( 'oui' , 'non' )

end program test1
```

fichier *somme.f90*

```
subroutine somme(x,y)

implicit none

real, intent(in) :: x,y
real :: s

s=x+y
print *, 'somme=' ,s

end subroutine somme
```

en entrée : 2 chaînes au lieu de 2 réels \Rightarrow erreur pas détectée à la compilation. A l’exécution, on a un résultat aberrant.

2. interface explicite avec module fichier *test2.f90*

```
program test2

use mod_somme

implicit none

call somme( 'oui' , 'non' )

end program test2
```

fichier *mod_somme.f90*

```
module mod_somme

implicit none

contains

subroutine somme(x,y)

implicit none

real , intent(in) :: x,y
real :: s

s=x+y
print *, 'somme=' , s

end subroutine somme

end module mod_somme
```

l'erreur est alors détectée à la compilation :

```
test2.f90:7.13:
```

```
call somme('oui','non')
      1
Error: Type mismatch in argument 'x' at (1); passed CHARACTER(1) to REAL(4)
```

- autres utilisations des modules
 - permettent de définir des constantes, des variables, des types dérivés que l'on peut utiliser dans différents sous-programmes sans les passer en argument.
- Exemple :
- fichier *mod_cte_math.f90*

```
module mod_cte_math

implicit none

real , parameter :: pi=3.141593

end module mod_cte_math
```

fichier *aire.f90*

```
program aire

use mod_cte_math

implicit none

real :: r,s

r=1.0
s=pi*r**2

end program aire
```

- permet aussi de passer des procédures en arguments, pour faire des fonctions de fonctions (voir section 5)
- conseil : utiliser les modules autant que possible.

4.5 Compilation séparée

comment compiler un programme quand on a des sous-programmes externes situés dans des fichiers différents ?

- compilation globale
 - utiliser la commande vue sec 2.9
 - gfortran -o nom_executable liste_fichiers_fortrans*
 - mais les modules doivent être compilés avant les sous-programmes utilisateurs. Ils doivent donc être placés avant dans la liste
 - inconvénient : une modification d'un fichier entraîne la re-compilation de tous les fichiers

- compilation séparée : 2 phases successives
 1. compilation (option `-c`) : pour tout fichier fortran (ici *fich.f90*) faire

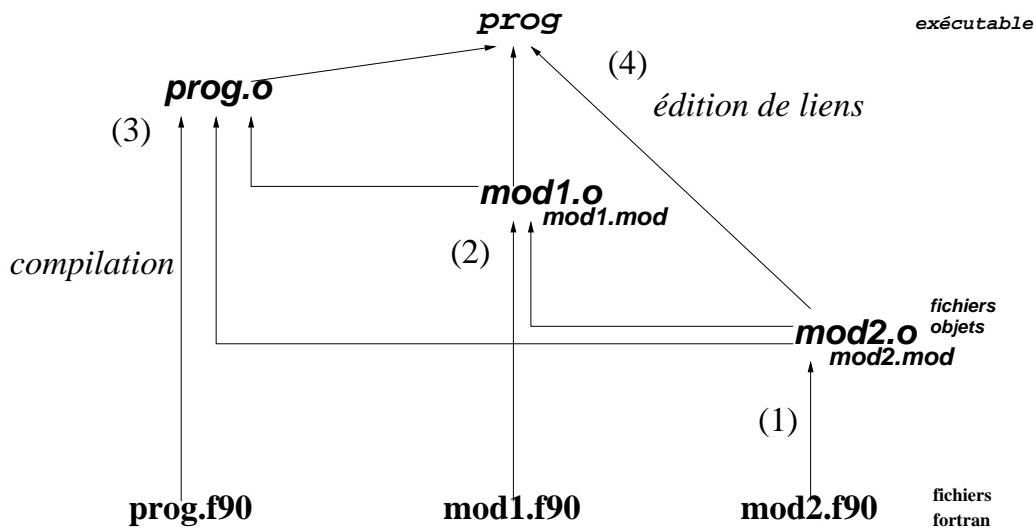
```
gfortran -c fich.f90
```

résultat :
 - création du fichier objet *fich.o* (de même nom que le fichier fortran mais d'extension `.o`)
 - si *fich.f90* contient le module **fich**, création du fichier *fich.mod* (de même nom que le module avec l'extension `.mod`). Contient l'interface du module, nécessaire pour compiler les programmes utilisateurs de **fich**
 - attention : compiler les modules avant les sous-programmes utilisateurs
 2. édition de liens (option `-o`) : les fichiers objets (`.o`) sont liés ensemble pour faire l'exécutable

```
gfortran -o nom_executable liste_fichiers_objets
```

- intérêt :
 - on ne recompile que les fichiers modifiés et les fichiers dépendants, et l'on refait l'édition de liens
 - outil complémentaire efficace pour la compilation séparée de gros programmes (plusieurs dizaines de fichiers) : la commande *make* (voir la fin de cette section)

- Exemple : programme fortran90 composé d'un programme principal **prog** qui utilise deux modules **mod1** et **mod2**. Le module **mod1** utilise le module **mod2**.



⇒ la compilation séparée est :

(1) compilation :

```
f90 -c mod2.f90
```

(2) compilation :

```
f90 -c mod1.f90
```

(3) compilation :

```
f90 -c prog.f90
```

(4) édition de liens :

```
f90 -o prog prog.o mod1.o mod2.o
```

⇒ en cas de modification de `prog`, alors seules la compilation de `prog.f90` puis l'édition de liens sont nécessaires (phases (3) et (4))

- utilisation de la commande `make` : le fichier suivant (nommé “Makefile”) contient les dépendances entre fichiers et les actions à exécuter pour mettre à jour l’exécutable :

```
# edition de liens
prog : prog.o mod1.o mod2.o
<TAB> gfortran -o prog prog.o mod1.o mod2.o

# compilation
prog.o : prog.f90 mod1.o mod2.o
<TAB> gfortran -c prog.f90
mod1.o : mod1.f90 mod2.o
<TAB> gfortran -c mod1.f90
mod2.o : mod2.f90
<TAB> gfortran -c mod2.f90
```

La commande unix `make` utilise ce fichier pour ne compiler que ce qui a été modifié : il suffit de taper cette commande dans le dossier contenant le programme.

Attention : la tabulation (visualisée ci-dessus par <TAB>) est indispensable.

- on peut paramétriser ce “Makefile” pour utiliser des options de compilation différentes ou pour “nettoyer” le dossier :

```

#paramètres
DEBUG=
COMPILE=gfortran $(DEBUG)

# edition de liens
prog : prog.o mod1.o mod2.o
<TAB> $(COMPILE) -o prog prog.o mod1.o mod2.o
<TAB> @echo compilation terminée

# compilation
prog.o : prog.f90 mod1.o mod2.o
<TAB> $(COMPILE) -c prog.f90
mod1.o : mod1.f90 mod2.o
<TAB> $(COMPILE) -c mod1.f90
mod2.o : mod2.f90
<TAB> $(COMPILE) -c mod2.f90

# destruction des fichiers objets et modules
detruire :
<TAB> rm -f *.o *.mod

```

- différents appels de `make` :
 - `make` : compilation normale
 - `make OPT=-g` : compilation en mode debug
 - `make OPT=-O2` : compilation optimisé de niveau 2
 - `make detruire` : efface les .o et les .mod
- Voir [4, 1] pour des détails.
- Remarque : pour les petits programmes, le makefile peut être généré à la main, mais pour les programmes plus longs, il peut être utile de le générer automatiquement, par exemple avec le script perl `mkmf`, voir :
<http://www.gfdl.noaa.gov/~vb/mkmf.html>

5 Utilisation avancée

- les concepts présentés ici sont plus complexes que les précédents
- à part le concept de précision des réels, les autres ne sont pas forcément nécessaires pour réaliser des programmes de calcul scientifique, sauf peut-être les pointeurs qui peuvent être indispensables dans certains cas
- cependant, ces concepts peuvent permettre de simplifier l'écriture des programmes

5.1 Types dérivés

- structures de données renfermant plusieurs types différents
- exemple : type **personne** contenant les nom, prénom, et age d'une personne :

```
type personne
  character(len=30) :: nom, prenom
  integer :: age
end type personne
```

on peut alors déclarer des variables de ce type, comme pour n'importe quel type

```
type(personne) :: etudiant1, etudiant2
type(personne), dimension(1:35) :: td
```

- les différentes variables d'un type dérivé sont appelées *champs*. On y accède avec le caractère %, et on peut utiliser le constructeur de même nom que le type pour les initialiser :

```
etudiant1=personne( 'Cesar' , 'Jules' , 57)
etudiant2=etudiant1
print*, etudiant2

do i=1,2
    print*, 'nom, prenom, age'
    read*, td(i)%nom, td(i)%prenom, td(i)%age
end do

print*, td(1)
```

- l'affectation etudiant2=etudiant1 recopie tous les champs.

5.2 Précision des réels

- par défaut, le type **real** est dit "simple précision", c.-à-d. codé sur 4 octets (=32 bits), avec 6 chiffres significatifs : ceci est souvent insuffisant pour faire des calcul scientifique précis ;
- il faut alors utiliser la représentation "double précision", c.-à-d. codée sur 8 octets (=64 bits), avec 15 chiffres significatifs \Rightarrow 3 possibilités :

→ type **double precision** :

```
double precision :: x
```

→ type **real** avec paramètre **kind** :

```
integer , parameter :: pr=kind(1.d0)
real(pr) :: x
```

(type réel de même précision que 1.d0, c.-à-d.
double précision)

→ ou

```
integer , parameter :: pr=selected_real_kind(15,3)
real(pr) :: x
```

(type réel avec 15 chiffres significatifs et exposant
à 3 chiffres, c.-à-d. double précision)

- la 3^epossibilité est la plus "portable" (donne la même précision quelle que soit la machine)

- les valeurs numériques s'écrivent différemment selon la précision choisie

valeur	real	double precision	real (pr)
1	1.	1.d0	1._pr
4,32	4.32	4.32d0	4.32_pr
$4,32 \times 10^{-4}$	4.32e-4	4.32d-4	4.32e-4_pr

- attention à la cohérence des types dans les affectations ou opérations :

```

integer :: n
double precision :: u
n = 1
u = n          ! -- u = 1.00000000000000
u = 1.2d0      ! -- u = 1.20000000000000
n = u          ! -- n = 1
u = 1.2        ! -- u = 1.2000004768372

```

- par prudence : ne pas mélanger les entiers et réels
- complexes : comme pour les réels :

```

complex :: z           ! --- simple p .
double complex :: z     ! --- double p .
pr=kind(1.d0)           ! -- ou
pr=selected_real_kind(15,3)
complex(pr) :: z       ! --- double p .

```

5.3 Fonctions de fonctions

- il s'agit simplement de créer des procédures dont les arguments peuvent être d'autres procédures
- exemple simple : écrire une fonction `integrale` qui pour toute fonction f et tout couple de réels a, b calcule une approximation de $\int_a^b f(x) dx$
- pour cela : le fonction `integrale` doit être mise dans un **module**, et le type de la fonction f et de ses arguments doivent être déclarés dans un bloc **interface** à l'intérieur de la fonction `integrale`
- attention : dans le programme utilisant la fonction `integrale`, la fonction f passée en argument est :
 - soit une fonction intrinsèque (comme `cos`, `log`, `exp`, `sqrt` ...), il faut alors la déclarer avec l'attribut **intrinsic** ;
 - soit définie dans un module `toto`, il suffit alors de déclarer l'utilisation de ce module (**use toto**) ;
 - soit définie dans un sous-programme externe non placé dans un module, il faut alors la déclarer avec l'attribut **external**.

– exemple :

```
module mod_integrale
    implicit none

contains

    real function integrale(f,a,b)
        implicit none

        ! --- arguments
        real , intent(in) :: a,b

        ! --- argument f : bloc interface
        interface
            real function f(x)
                real , intent(in) :: x
            end function f
        end interface

        ! --- var locales
        integer :: n,i
        real :: h

        n=100 ; h=(b-a)/n
        integrale=0.
        do i=0,n-1
            integrale=integrale+h*f(i*h)
        end do

    end function integrale

end module mod_integrale
```

- utilisation de cette fonction dans un programme :

```
program prog_integrale

use mod_integrale
use mod_creneau
implicit none

real :: y1,y2
real , intrinsic :: exp

y1=integrale(exp,0.,1.)
y2=integrale(creneau,-1.,1.)
print *,y1,y2

end program prog_integrale
```

```
module mod_creneau
implicit none
contains
  real function creneau(x)
    implicit none
    real , intent(in) :: x

    if (x<0.0.and.x>1.0) then
      creneau=0.
    else
      creneau=1.
    end if
  end function creneau
end module mod_creneau
```

5.4 Interface générique

- idée : on veut appliquer une même procédure à des arguments de type différents. Exemple : `abs(x)` renvoie $|x|$ si x est réel et $\sqrt{a^2 + b^2}$ si $x = a + ib$ est complexe.
- principe : dans un module, on écrit plusieurs procédures adaptées à chaque type d'arguments, et on les regroupe sous un même nom “générique”. C'est le compilateur qui décide quelle procédure utiliser selon le type des arguments.
- exemple : on veut une fonction `exponentielle` qui renvoie $\exp(x)$ si x est réel, et $\exp(z) = \exp(a)(\cos b + i \sin b)$ si $z = a + ib$ est complexe

```

module mod_exponentielle
  implicit none
  interface exponentielle
    module procedure rexp, zexp
  end interface
contains
  function rexp(x) result(y)
    implicit none
    real, intent(in) :: x
    real :: y

    y=exp(x)
  end function rexp

  function zexp(z) result(y)
    implicit none
    complex, intent(in) :: z
    complex :: y
    real :: a,b

    a=real(z); b=aimag(z)
    y=cplx(cos(b),sin(b))*exp(a)
  end function zexp

end module mod_exponentielle

```

- méthode : deux fonctions **rexp** et **zexp** stockées dans le module **mod_exponentielle**. La fonction générique **exponentielle** est créée avec le bloc **interface** et l’instruction **module procedure** située au tout début du module.

- ensuite, dans un programme principal, on utilise la même fonction exponentielle, que l'argument soit réel ou complexe :

```
program generique
  use mod_exponentielle
  implicit none

  real :: x
  complex :: z

  x=0. ; z=cmplx(0.,3.1416)

  print*, 'exp(x)=',exponentielle(x)
  print*, 'exp(z)=',exponentielle(z)

end program generique
```

5.5 Cr eation de nouveaux op rateurs

- exemple : on a une fonction `rptens` qui avec deux vecteurs r eels v_1 et v_2 en entr ee renvoie le produit tensoriel $w = v_1 \otimes v_2$:

```
w=rptens(v1,v2)
```

- on voudrait utiliser cette fonction sous forme d'un *op rateur* `ptens` :

```
w=v1.ptens.v2
```

- on va utiliser un **module**, un bloc **interface operator** et l'instruction **module procedure** :

```

module mod_ptens

  implicit none
  interface operator(.ptens.)
    module procedure rptens
  end interface
contains

  function rptens(v1,v2) result(w)
    implicit none
    real, dimension(:) , intent(in) :: v1,v2
    real, dimension(size(v1),size(v2)) :: w
    integer :: i,j

    do i=1,size(v1)
      do j=1,size(v1)
        w(i,j)=v1(i)*v2(j)
      end do
    end do

  end function rptens

end module mod_ptens

```

- les deux points autour de .ptens. sont obligatoires pour définir un opérateur
- dans le programme principal utilisant le module mod_ptens, on pourra utiliser l'opérateur ptens ainsi :

w=v1.ptens.v2

- on peut associer plusieurs fonctions à un même opérateur (comme avec les interfaces génériques, section 5.4) : pour que `.ptens.` marche aussi avec les complexes, rajouter une fonction `zptens` dans le module, et ajouter le nom `zptens` à la suite de `rptens` dans l'instruction **module procedure**
- on peut aussi vouloir redéfinir un opérateur existant comme `*, +, -, /` etc. Cela est possible, mais un peu plus compliqué (il faut que le nouvel opérateur et l'ancien diffèrent par le type de leurs arguments). Dans l'exemple précédent, on ne pourrait pas redéfinir `*` par l'opérateur `ptens` car `*` est déjà défini pour les tableaux.

- exemple de surcharge de + : pour concaténer des chaînes comme l'opérateur //, en supprimant des blancs. Remarquer que + garde sa signification pour les autres types.

```
program surcharge

use mod_add_chaine
implicit none

character(len=20) :: c1,c2,c

c1='bon' ; c2='jour'
c=c1+c2
print*,c
print*, 'hello da'+ ' rling'
print*,5+3

end program surcharge
```

```
module mod_add_chaine

implicit none
interface operator(+)
    module procedure add_chaine
end interface

contains

function add_chaine(c1,c2) result(c)

    implicit none

    character(len=*) , intent(in) :: c1,c2
    character(len=len(c1)+len(c2)) :: c
    c=trim(adjustl(c1))//trim(adjustl(c2))

end function add_chaine

end module mod_add_chaine
```

5.6 Ressources privées, publiques, semi-privées

- quand un programme utilise un module, il a accès à toutes les variables du module, et tous ses sous-programmes. Cela peut parfois provoquer des conflits avec les propres variables du programme si les noms sont identiques
- exemple : un programmeur veut utiliser un module servant à calculer des normes de matrices avec la fonction `norme` pour son programme. Ce module contient beaucoup d'autres fonctions auxiliaires utilisées par la fonction `norme`. Le programmeur ne connaît pas ces autres fonctions (il n'a accès qu'au fichier objet par exemple). Dans ce cas, les fonctions auxiliaires ne devraient pas être visibles à l'extérieur du module, car sinon, l'utilisateur n'aurait par exemple pas le droit de créer des fonctions portant le même nom, alors qu'il ignore leur existence !
- on peut alors utiliser les fonctions **private** (privé), **public** et les attributs correspondants **private** et **public** pour limiter l'accès à certaines variables depuis l'extérieur du module
- voir la référence [3] pour plus de détails.

5.7 Pointeurs

- la structure de pointeur est fondamentale en langage C et C++. En fortran 90, une structure similaire existe, mais n'est pas d'un intérêt évident pour le calcul scientifique
- dans la norme fortran 95, elle était indispensable pour définir des tableaux allouables à l'intérieur de types dérivés (comme par exemple pour définir des tableaux de tableaux). Mais depuis la norme fortran 2000, cela n'est plus nécessaire.
- la notion de pointeur en fortran 90 ne sera donc pas étudiée plus en détail dans ce cours

Références

- [1] MANUELS GNU. *manual make*.
<http://www.gnu.org/software/make/manual/make.html>.
- [2] MANUELS GNU. *Manuel de ddd, interface graphique de debugger*.
http://www.gnu.org/s/ddd/manual/html_mono/ddd.html.
- [3] IDRIS. *Cours IDRIS Fortran*, 2004.
http://www.idris.fr/data/cours/lang/f90/F90_cours.html.
- [4] Luc Mieussens. *Cours unix-shell-makefile*. Institut de Mathématiques - Université de Bordeaux, 2004.
http://www.math.u-bordeaux1.fr/~mieussen/PAGE_WEB/ENSEIGNEMENT/unix_shell.pdf.
- [5] Luc Mieussens. *Petit guide pour Emacs*. Institut de Mathématiques - Université de Bordeaux, 2004.
http://www.math.u-bordeaux1.fr/~mieussen/PAGE_WEB/ENSEIGNEMENT/guide_emacs_version_140904.pdf.

Liste de quelques fonctions intrinsèques usuelles en Fortran 90

*adapté des cours de l'IDRIS, disponibles sur
http://www.idris.fr/data/cours/lang/f90/F90_cours.html*

On donne la signification de la commande, puis, dans un exemple, la syntaxe à utiliser, suivie après le signe ' \mapsto ' de la valeur retournée en sortie.

1 Fonctions intrinsèques sur les entiers, réels, complexes, extensibles aux tableaux

- **ABS** : retourne la valeur absolue de son argument. Pour un complexe, retourne sa norme : $\sqrt{x^2 + y^2}$.

ABS(-1) \mapsto 1
ABS(-1.5) \mapsto 1.5; **ABS**((3.,4.)) \mapsto 5.0

- **ACOS** : retourne l'arc cosinus en radians de son argument réel.

ACOS(0.54030231) \mapsto 1.0

- **AIMAG** : retourne la partie imaginaire du complexe passé en argument.

AIMAG((2.,3.)) \mapsto 3.0

- **AINT** : tronque le réel passé en argument

AINT(2.783) \mapsto 2.0; **AINT**(-2.783) \mapsto -2.0

- **ANINT** : retourne, sous forme d'un réel, l'entier le plus proche du réel transmis.

ANINT(2.783) \mapsto 3.0
ANINT(-2.783) \mapsto -3.0

- **ASIN** : retourne l'arc sinus en radians de son argument réel.

ASIN(0.84147098) \mapsto 1.0

- **ATAN** : retourne l'arc tangente en radians de son argument réel.

ATAN(1.5574077) \mapsto 1.0

- **CEILING** : retourne l'entier immédiatement supérieur au réel transmis en argument.

CEILING(3.7) \mapsto 4
CEILING(-3.7) \mapsto -3

- **CMPLX** : retourne un complexe dont les parties réelle et imaginaire sont transmises en argument.

CMPLX(-3.) \mapsto -3.0+0.i; **CMPLX**(2,4.) \mapsto 2.0+4.0i

- **CONJG** : retourne le complexe conjugué de celui passé en argument.

CONJG((-3.0,4.0)) \mapsto -3.0-4.0i

- **COS** : retourne le cosinus de l'angle passé en argument (exprimé en radians).

COS(1.0) \mapsto 0.54030231

- **COSH** : retourne le cosinus hyperbolique.

COSH (1.0) \mapsto 1.5430806

- **DBLE** : convertit en double précision l’argument transmis.
- **EXP** : retourne l’exponentiel de l’argument transmis.

EXP (1.0) \mapsto 2.7182818

- **FLOOR** : retourne l’entier immédiatement inférieur au réel transmis en argument.

FLOOR (3.7) \mapsto 3 ; **FLOOR** (-3.7) \mapsto -4

- **INT** : convertit en entier l’argument transmis.

INT (-3.7) \mapsto -3

INT (9.1/4.0) \mapsto 2

- **LOG** : retourne le logarithme népérien de l’argument transmis.

LOG (2.7182818) \mapsto 1.0

LOG (10.0) \mapsto 2.3025851

- **LOG10** : retourne le logarithme décimal de l’argument transmis.

LOG10 (10.0) \mapsto 1.0

LOG10 (10.E10) \mapsto 11.0

- **MAX** : retourne le maximum des nombres passés en argument.

MAX (-9.0, 7.0, 2.0) \mapsto 7.0

- **MIN** : retourne le minimum des nombres passés en argument.

MIN (-9.0, 7.0, 2.0) \mapsto -9.0

- **MOD** : retourne le reste de la division effectuée à l’aide des deux arguments fournis.

MOD (3.0, 2.0) \mapsto 1.0

MOD (-8,5) \mapsto -3

- **REAL** : convertit en réel l’argument transmis.

REAL (3) \mapsto 3.0

- **SIGN** : retourne le nombre dont la valeur absolue est celle du premier argument et le signe celui du deuxième.

SIGN (-3.0, 2.0) \mapsto 3.0

- **SIN** : retourne le sinus de l’angle passé en argument (exprimé en radians).

SIN (1.0) \mapsto 0.84147098

- **SINH** : retourne le sinus hyperbolique.

SINH (1.0) \mapsto 1.1752012

- **SQRT** : retourne la racine carré de son argument.

SQRT (5.0) \mapsto 2.236068010

- **TAN** : retourne la tangente de l’angle passé en argument (exprimé en radians).

TAN (1.0) \mapsto 1.5574077

- **TANH** : retourne la tangente hyperbolique.

TANH (1.0) \mapsto 0.76159416

2 Fonctions intrinsèques pour le temps de calcul

- **CPU_TIME(t)** (Norme 95) retourne dans le réel t le temps CPU en secondes (ou réel < 0 si indisponible). Par différence entre deux appels, il permet d'évaluer la consommation CPU d'une section de code.

```
call cpu_time(t1)
...
call cpu_time(t2)
print*, 'temps_CPU=' ,t2-t1
```

3 Fonctions intrinsèques pour les chaînes de caractères

- **ADJUSTL** : cadre à gauche la chaîne passée en argument : supprime les blancs en tête ; complète à droite par des blancs.

```
ADJUSTL( '_____Fortran' ) → 'Fortran_____'
```

- **ADJUSTR** : cadre à droite la chaîne passée en argument : supprime les blancs en fin ; complète à gauche par des blancs.

```
ADJUSTR( 'Fortran_____' ) → '_____Fortran'
```

- **LEN** : retourne la longueur de la chaîne de caractères transmise en argument.

```
CHARACTER(len=10) CH; LEN(CH) → 10
```

- **LEN_TRIM** : retourne la longueur de la chaîne de caractères transmise en argument sans considérer les blancs de fin.

```
LEN_TRIM( '____FORTRAN____' ) → 9
```

```
LEN_TRIM( '____' ) → 0
```

- **LGE** : compare les deux chaînes de caractères transmises en argument : retourne .true. si la première chaîne est supérieure ou égale à la deuxième, .false. sinon.

```
LGE( 'MANET' , 'MONET' ) → .false.
```

```
LGE( 'MANET' , 'MANET' ) → .true.
```

- **LGT** : compare les deux chaînes de caractères transmises en argument : retourne .true. si la première chaîne est supérieure strictement à la deuxième, .false. sinon.

```
LGT( 'MANET' , 'MANET' ) → .false.
```

- **LLE** : compare les deux chaînes de caractères transmises en argument : retourne .true. si la première chaîne est inférieure ou égale à la deuxième, .false. sinon.

```
LLE( 'MANET' , 'MONET' ) → .true.
```

```
LLE( 'MANET' , 'MANET' ) → .true.
```

- **LLT** : compare les deux chaînes de caractères transmises en argument : retourne .true. si la première chaîne est inférieure strictement à la deuxième, .false. sinon.

```
LLT( 'MANET' , 'MANET' ) → .false.
```

- **TRIM** : retourne la chaîne de caractères transmise débarrassée de ses blancs de fin.

```
TRIM( 'PICASSO_____') → 'PICASSO'
```

- // : opérateur pour concaténer deux chaînes.

```
'bon' // 'jour' → 'bonjour'
```

4 Fonctions et subroutines intrinsèques tableaux

4.1 interrogation

- `SIZE(array[,dim])` retourne la taille (ou l'étendue de la dimension indiquée via `dim`) du tableau passé en argument.

```
integer, dimension(-2:27,0:49) :: t
SIZE(t)    ↪ 1500
SIZE(t,dim=1) ↪ 30
```

- `MAXLOC(tab)` et `MINLOC(tab)` retournent, pour le tableau `tab`, l'emplacement de l'élément maximum/minimum (si le tableau est multidimensionnel, le résultat est un vecteur dont les éléments sont les indices de l'élément dans `tab`)

```
integer, dimension(-2:27,0:49) :: t
integer, dimension(1:2) :: ind_elem_max, ind_elem_min
ind_elem_max=maxloc(t)
ind_elem_min=minloc(t)
```

4.2 Réduction

Selon que `DIM` est absent ou présent, toutes ces fonctions retournent soit un scalaire soit un tableau de rang $n-1$ en désignant par n le rang du tableau passé en premier argument.

La plupart de ces expressions acceptent un “masque” en paramètre, qui permet de n’appliquer les fonctions qu’à certains éléments (voir le cours de l’IDRIS de Fortran 95)

- `COUNT(verif)` : `verif` est une expression logique sur deux tableaux conformes, `count` compte le nombre de fois où cette expression est vérifiée par deux éléments

Soit A | 1 3 5 |
| 2 4 6 |

Décompte global des valeurs de $A > 2$:

```
COUNT(A>3) ↪ 4
```

- `MAXVAL(array)` et `MINVAL(array)`

Soit A | 1 3 5 |
| 2 4 6 |
 $\maxval(A) \mapsto 6$
 $\minval(A) \mapsto 1$

- `PRODUCT(tab)` renvoie le produit de tous les éléments du tableau `tab`.

```
PRODUCT((/ 2,5,-6 /)) ↪ -60
```

- `SUM(tab)` renvoie la somme de tous les éléments du tableau `tab`.

```
SUM((/ 2,5,-6 /)) ↪ 1
```

4.3 multiplication

- `DOT_PRODUCT(vector_a,vector_b)` retourne le produit scalaire des deux vecteurs passés en argument (produit hermitien si `vector_a` est complexe)

```
v1 =(/ 2,-3,-1 /) ; v2 =(/ 6,3,3 /)
DOT_PRODUCT(v1,v2) ↪ 0
```

- **MATMUL(matrix_a, matrix_b)** effectue le produit matriciel de deux matrices ou d'une matrice et d'un vecteur passés en argument. Soit la matrice $A = \begin{pmatrix} 3 & -6 & -1 \\ 2 & 3 & 1 \\ -1 & -2 & 4 \end{pmatrix}$ et le vecteur $v = \begin{pmatrix} 2 \\ -4 \\ 1 \end{pmatrix}$
- MATMUL (A , V)**
- $$\mapsto v = \begin{pmatrix} \frac{29}{7} \\ 10 \end{pmatrix}.$$

4.4 transformation

- **TRANSPOSE(mat)** renvoie la transposée de la matrice **mat** (tableau bidimensionnel)